WOBURN CHALLENGE

2015-16 Online Round 1

Solutions

Problem J1: Do You Know Your Woburn Colours?

This is a straightforward task solved using an if..else statement. It tests the competitor's basic abilities in input/output, comparing strings, and simple logic. Given two of three strings in the set of {"RED", "BLUE", "WHITE"}, we are to identify the other one that was not mentioned.

Initially, one may be tempted to try and code for every possible case of the input (there are 6 combinations) by checking the two inputted strings against ways to choose exactly two of the three colours, taking into account the fact that order matters. This is a feasible way to do it. However, it leads to a solution that is more complicated than necessary. Every second matters in a timed contest, so the less code we have to write the better. To simplify the 6 cases to only 3 cases without added complexity, we can just check which colour *isn't* in the input. That is, if neither of the input strings compare equal to a particular colour, then this colour must be the one we output.

Official Solution (C++)

```
#include <iostream>
using namespace std;

string s1, s2;

int main() {
  cin >> s1 >> s2;
  if (s1 != "RED" && s2 != "RED") {
    cout << "RED" << endl;
  } else if (s1 != "BLUE" && s2 != "BLUE") {
    cout << "BLUE" << endl;
  } else {
    cout << "WHITE" << endl;
  }
  return 0;
}
```

Problem J2: Grouping Recruits

This is a simple math problem. We want to divide *N* students up into *M* groups such that the difference between the sizes of the largest and smallest groups is minimized. An analogous situation to imagine is the process of dealing out cards one-by-one to players seated around a table – players get as equal number of cards as possible. We can choose to approach this problem using simulation or mathematics.

We can simulate the assignment of students by keeping a counter of how many students have yet to be distributed (initialized to *N*), as well storing the number of students currently in each group as an array (each array cell initialized to 0). Then, we would loop upwards through each group (resetting our loop variable to the index of the first group when we reach the last one), decrementing our counter for each iteration whilst incrementing the value at the current index of the array, until the number of students have reached 0. Finally, we would go back to the array and count the frequencies of each group size and report the answer.

This method would have worked in the contest, but it should be obvious that it is overcomplicated to write out in code. It's always a good idea to make our code as simple as possible to avoid the introduction of bugs that can take time to find and fix. This is especially true in timed contests, where we should always think a bit more about better alternate solutions before plunging straight into code. Furthermore, *N* and *M* happened to have small bounds of 100 for this problem. Had the bounds been greater (say, 1 billion), our current method would likely not have executed under the 2 second time limit. We can reach a much more elegant and efficient mathematical solution by making some simple observations.

The first observation is that there are only two possibilities for the answer: either the students are evenly distributed (all group sizes are the same) or some groups will have at most 1 more recruit than others (there are exactly two group sizes). The reason for this should be intuitive – we know that there definitely exists a way to make *M* groups of equal sizes which are as big as possible, simply by having some students left over. The number of students left over must be smaller than *M*, because otherwise we would create even bigger evenly-sized groups by taking *M* students from the leftovers to distribute one-each amongst the groups (contradicting how we assumed the groups were as big as possible). Intuition is an important skill to have during programming contests because we do not have to prove anything rigorously in order to implement it. As long as we're intuitively convinced that it is right, we can go for the answer.

With that in mind, we have to actually come up with formulas. Our formulas will require the modulo operation (mod) which takes the remainder after division. 5 mod 2 is equal to 1, for example. It is available in nearly all programming languages (the operator is the % sign in both C++ and Python). So, let's consider the two cases separately:

- To check if we can evenly divide *N* students amongst *M* groups, we just check if the remainder of *N* divided by *M* is 0. If *N* mod *M* is 0, we know that there are *M* groups of size *N*/*M* each.
- Otherwise, it must be that there are 2 group sizes. Intuitively, we know that some groups will have size *N*/*M*, rounded down (call these type 1 groups). Since we know that we'll have to distribute the leftover students, some groups will have a size that's exactly 1 more than the value of *N*/*M* rounded down (call these type 2 groups).
  How many type 2 groups are there? Well that's exactly the number of leftover students there are (*N* mod *M*, since we're distributing 1 leftover student to each type 2 group)! However many type 2 groups there are, the number of type 1 groups must be be exactly *M* minus that number since there are *M* groups total.

The above idea is implemented in the official solutions below. Note that the '//' operator in Python performs integer division (division rounded down), as does the '/' operator in C++, if the operands are integer types.

Official Solution (C++)

```
#include <iostream>
using namespace std;

int N, M;

int main() {
  cin >> N >> M;
  if (N % M == 0) {
    cout << M << " group(s) of " << (N / M) << endl;
  } else {
    cout << (M - N % M) << " group(s) of " << (N / M) << endl;
    cout << (N % M) << " group(s) of " << (N / M + 1) << endl;
  }
  return 0;
}
```

Problem J3: Jazz Concert

This is a basic implementation problem. We want to make the concert as long as possible by picking two of the *N* songs to repeat. Obviously, we should always pick two of the longest songs. In the end, we add up the lengths of all the songs in the input, add again the sum of the lengths of the two longest songs, and finally add 10 for the time of the intermission.

How do we find the lengths of the two longest songs? One way is to store all the song lengths into an array and sort the array. If we sort the array in descending order, the two longest lengths will be at the first and second indices the array (since it's guaranteed that there are at least two songs). If we sort the array in ascending order, the two longest lengths will be at the last and second last indices. Many languages come up with built-in sort functions, so it is a viable way to approach the problem.

However, using arrays and sorting may seem like overkill for just finding the top two elements. As it turns out, we can also solve the problem by processing each song length as we read them in – there is no need to store all of them at once! We keep two variables: the top value (maximum length) we've encountered so far and the second value (next greatest length) we've encountered so far. For every new *T* value we read in, we add it to the running total and update the two variables as follows. If *T* is greater than the top value, we set the second value to the top value and then update the top value to *T*. It is important to do it in this order, otherwise we would be setting the second value to *T* and not the old top value. At the end, we add 10 along with these two variables to the running total and output the result.

The following official solutions implement the latter method without an array.

Official Solution (C++)

```cpp
#include <iostream>
using namespace std;

int N, T, total = 10, first = 0, second = 0;

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> T;
    if (T > first) {
      second = first;
      first = T;
    } else if (T > second) {
      second = T;
    }
    total += T;
  }
  cout << total + first + second << endl;
  return 0;
}
```

Problem J4: Trip Budgeting

This question is the trickiest of the entire junior contest, and tests the competitor's ability to perform a brute force search. Given N attractions, we would like to know the subset of attractions we select in order to minimize the cost they incur. Now, notice that if we are given a set of attractions, we can very easily follow the instructions to calculate out how much the attraction will cost (as well as the level of excitement it generates). After solving the last problem, it should seem like basic busywork for us to calculate the sum and max of a bunch of values. So the crux of the question is really "which attractions do we pick?" In such contests, it's always a good idea to keep it simple. A simple, inefficient idea that passes under the time limit is always better than an advanced, efficient idea which is hard to implement. This is why we should always first ask ourselves whether we can "brute-force" it (try all possibilities). That is, we want to generate every possible subset of attractions and then compute the excitement level and cost of each one, comparing the ones with at least an excitement of E_{min} each other to find the best cost.

To determine whether an idea is feasible (in this case, fast enough to pass under the time limit), we will have to introduce the concept of asymptotic analysis, which is a method to describe how rapidly an algorithm slows down as the input grows in size. Big O notation is used to describe the rate of growth of a function as a function of the input size. These concepts are key in analyzing algorithms and understanding them is critical for solving harder contest problems. There are many great articles on these topics that may be found online, so we will not explain it here.

In this case, we want to know how quickly our brute-force algorithm will slow down as the number of attractions increase. So for every attraction, we have two choices: either choose to attend the attraction or don't. Since we have 2 possible choices for each of N total times, the number of total ways to pick out attractions is 2^N. Since for every way we pick out a configuration, we also have to calculate the excitement level as well as the costs by looping over the N attractions, we have to multiply the number of configurations by the number of "steps" we take per configuration to obtain the total number of "steps" that our program can take in the worst case. In big O notation, we can say that our program runs in $O(2^N \times N)$. Actually plugging numbers into this for the worst case (largest possible N, which is 20), we get $2^{20} \times 20 = 20971520$ (roughly 20 million) possible ways to pick out the attractions. A good rule of thumb for programming contests is to have at most 30 million "steps" per second of time limit. Since we have 2 seconds of time limit, 20 million steps in our brute-force algorithm should comfortably pass.

After confirming that our idea is efficient enough, the next step is to implement it. There are two approaches we will explore: recursion (backtracking) and bitmasks.

Most competitors have decided to solve the problem using recursion. The idea is to enumerate every possible subset of the N attractions using recursive "backtracking", and after we've done it for all N attractions (reaching the base case), we calculate the excitements and costs and update our answer accordingly. Once again, you can look up articles online to better understand the concept. Many recursive approaches submitted during the contest apply certain forms of optimization involving pruning out configurations before we've generated all N choices, but our analysis above proved that this was not necessary. As an aside, passing arrays as arguments to functions (especially throwing them around in recursive ones) is a common cause of confusion amongst many programmers (passing by reference vs. value works differently across languages, and programmers may not be familiar with these concepts), so it's good avoid altogether and try to handle everything globally.

All we need is to keep a global array of N Boolean values where each index stores whether we've decided to keep a particular attraction and a single integer as the parameter of our recursive function (the current attraction which we are making a decision for). At each step of the recursion, we will mark the current attraction as "attended" in the global array, then let the next layer of recursion handle the next attraction. Then in the same step we mark the attraction as "not attended", and let the next layer of recursion handle the next attraction. At the start of the function we check if all N attractions have been decided. If so, we calculate and update our answer before cutting off the recursion. This solution is implemented below.

Official Solution: Recursive Backtracking (C++)

```
#include <iostream>
using namespace std;

const int MAXN = 20;

int Emin, Tbase, Hbase, Fbase, ans = 1000000000;
int N, E[MAXN], T[MAXN], H[MAXN], F[MAXN];

bool attend[MAXN];

void rec(int curr) {
  if (curr == N) {
    int e = 0, t = Tbase, h = Hbase, f = Fbase;
    for (int i = 0; i < N; i++) {
      if (attend[i]) {
        e += E[i];
        t += T[i];
        h = max(h, H[i]);
        f = max(0, f - F[i]);
      }
    }
    int cost = t + h + f;
    if (e >= Emin) {
      ans = min(ans, cost);
    }
    return; //we're done
  }
  attend[curr] = true;
  rec(curr + 1); //attend current attraction
  attend[curr] = false;
  rec(curr + 1); //don't attend current attraction
}

int main() {
  cin >> Emin >> Tbase >> Hbase >> Fbase >> N;
  for (int i = 0; i < N; i++) {
    cin >> E[i] >> T[i] >> H[i] >> F[i];
    attend[i] = false;
  }
  rec(0);
  cout << ans << endl;
  return 0;
}
```

Alternatively, we can make the brute-force even simpler with a trick using "bitmasks" and binary numbers. Every integer has a base-2 (binary) representation, where it is simply a string of 0's and 1's. For N binary digits, it just so happens that there are 2^N distinct binary numbers that can be represented (with the lowest number being a string of 0's in base-2 and the highest being a string of 1's). By the basic properties of binary numbers, it is actually true that *all* possible configuration of 0's and 1's using exactly N digits will be represented. Our Boolean array can actually be stored in a single 32-bit integer, with 12 bits left to spare! If the i-th binary digit of a number is 1, we will say that we attend attraction i, otherwise if it's a 0, we say that we don't attend it. By simply looping a number (which we will call a "mask") upwards from 0 to $2^N - 1$, we will have enumerated all possible subsets of the N things. Now, we could convert each integer to a binary string using base-conversion algorithms and then access each bit, but this misses the beauty of bitmasks. Nearly all programming languages support bitwise operations of AND, OR, NOT, and shifts on integers making the accessing of individual bits really easy and efficient. Given our mask that is really just an integer (call it *M*), the only operation we care about is querying the status of a bit (say at the *i*-th position of the binary representation). The operation we need is `M and (1 leftshift i)` (or equivalently, `(M leftshift i) and 1`). If the bit is non-zero, then we know the *i*-th bit is set (and so we will consider the *i*-th attraction part of our current selected subset), otherwise we consider the attraction not to have been selected. More information can be found here.

Now why is this seemingly overcomplicated method advantageous to recursion? First, recursion is done by your computer with an internal call-stack which can make your program less efficient (not asymptotically as we've discussed above, but by a constant factor). Avoiding it and just sticking to loops may improve your program's performance significantly. But more importantly, bitmasks often make solutions ridiculously short and sweet. The following is a solution using bitmasks to demonstrate this. Note that `<<` is the left shift operator in C++, and `(1 << N)` results in a number that is 2^N (so looping while we're less than `1 << N` means we're looping up to and including $2^N - 1$).

Official Solution: Bitmasks (C++)

```
#include <algorithm>
#include <iostream>
using namespace std;

const int MAXN = 20;

int Emin, Tbase, Hbase, Fbase, N, ans = 1000000000;
int E[MAXN], T[MAXN], H[MAXN], F[MAXN];

int main() {
  cin >> Emin >> Tbase >> Hbase >> Fbase >> N;
  for (int i = 0; i < N; i++) {
    cin >> E[i] >> T[i] >> H[i] >> F[i];
  }
  for (int mask = 0; mask < (1 << N); mask++) {
    int e = 0, t = Tbase, h = Hbase, f = Fbase;
    for (int i = 0; i < N; i++) {
      if ((mask >> i) & 1) { //if the i-th bit is set
        e += E[i];
        t += T[i];
        h = max(h, H[i]);
        f = max(0, f - F[i]);
      }
    }
    int cost = t + h + f;
    if (e >= Emin)
      ans = min(ans, cost);
  }
  cout << ans << endl;
  return 0;
}
```

Problem S1: Woburn Workload

Given a bunch of due dates and durations, we must decide which assignments to focus on (and when) in order to minimize the penalty marks. Observe that penalty marks for any assignment is directly proportional to the time we spend *not* working on the assignment, regardless of which assignment is in question. As long as we spend the longest possible total time working, we spend the shortest possible total time *not* working, and in turn will incur the fewest possible total penalty marks.

Next, notice that since we can only work on one assignment at a time, we can greedily choose to work assignments that will contribute the most time to our total working time. Intuitively knowing when a greedy algorithm works is a good skill to have for programming contests, since we don't have to rigorously prove anything to be able to implement it. A good indicator that a greedy algorithm will work is the fact that by being greedy with respect to one property, we are not sacrificing anything else. In this case, every assignment carries the same rate of penalty per unit of time (maximizing time worked and minimizing penalty go hand-in-hand), so we are losing nothing by simply being greedy in terms of maximizing time worked. So, the greedy algorithm we apply is as follows:

- Read the assignments one by one and keep a running variable storing the "position" we are on our timeline. Initialize this variable to time 0.
- At any point in time, *position* will store the point in time that we've just decided to stop working on the most recent assignment (either because we've worked on it for all T_i minutes to produce the best work, or because we've gone past the due date and gave up, in which case *position* will be equal to the due date).
- Whenever an assignment comes up, start working the assignment immediately from our current position. In other words, get on working through all of the new assignment's minutes immediately at the current time.
- However, it might not be possible that we can do it before the assignment's due date. So, if *position* is greater than the due date, we set it equal to the due date and calculate the penalty to add to the answer.
- At this point, we are ready to handle the next assignment.

By studying the algorithm on some examples, you should be able to convince yourself that through this method, we're doing our absolute best to work on assignments for the longest possible time overall. If at any *position* (we have nothing to do for the moment) we are given an assignment that we can work on for all T_i minutes and still not go past its due date, we obviously should do it all. However, if we should decide between sticking to one assignment that we're already working on versus switching to a new assignment, we know that it's obviously better to move on to the most recent assignment that was inputted because the due date is later.

Official Solution (C++)

```cpp
#include <iostream>
using namespace std;

int N, M, T, position = 0, ans = 0;

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> M >> T;
    position += T;
    if (position > M) {
      ans += position - M;
      position = M;
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem S2: Wildcat Wildcards

This may have been a deceptively simple problem to many competitors. After reading it, you may even believe that you could solve all valid inputs by hand, since there are only 11 numbers to deal with. We want to optimally assign the 3 types of wildcards so that the *minimum* quantity across all of the 8 letter cards is maximized. In the cases worth 10% of marks with no wildcards to assign, this is as simple as finding the minimum of all of the quantities (the number of signs we can make is limited by the minimum of each quantity, since the letters are distinct).

One intuitive way to approach this problem is by repeatedly taking the smallest quantity across all the 8 letter cards, and trying to raise it to the second smallest quantity across the 8 cards through assigning wildcards. We prioritize taking vowel and consonant specific wildcards first, because those are more limiting. When we have no more letters to distribute, we go through all of them and take the minimum value. The catch is when we end up with two or more cards all having the smallest quantity. Then, we would have to resort to some trickier decisions and case work. Going down this path, we'll probably reach a point where our code becomes much more complicated than what we bargained for.

To obtain a simpler solution, we can start by pointing out a rather obvious fact that if it's possible to make N signs, then it's also possible to make any number of signs less than N (but not greater). Rather than trying to answer the question "what is the max number of signs we can make?", note that it is much easier to answer the question "*can* we make N signs?" The latter question is answered by simply assigning all available wildcards to try to make every quantity equal N. Once again, we should assign vowel and consonant wildcards before the generic wildcards. Since there are only 11 types of cards overall, we can safely assume that this "check" operation has $O(1)$ running time.

A good (but not good enough) way to move forward from here is to loop N upwards starting from 0, at each iteration asking ourselves whether N signs could be made using the wildcard assignment algorithm we've discussed above. Immediately when we reach an N for which we find N signs cannot be made, output $N - 1$ (the largest possible number for which that number of signs could be made). This is a good start, but since the answer can be over 10^8 (100 million), it is not good enough to pass in time for all of the test cases (a good rule of thumb is 30 million steps per second of time limit).

To solve the problem completely, we can use binary search. Binary search is classically used for finding the position of a number in a sorted array, but it can be used just as well in these discrete situations. In this case, we have a Boolean function which returns true up to a certain point, and false for everything after. Binary search can be used to pinpoint the exact position where function switches from true to false in $O(\log N)$ time, where N is the search space. This article is a good explanation of discrete binary searching. By binary searching on the value of N with our Boolean function being whether we can make N signs, we will be able to solve the problem in a very efficient manner. The following program implements this solution.

Official Solution (C++)

```cpp
#include <iostream>
using namespace std;

int v[2], c[6], wildv, wildc, wildany;

// Can we construct n signs with the letters we have?
bool can_make(int n) {
  int v_needed = 0, c_needed = 0;
  for (int i = 0; i < 2; i++)
    if (v[i] < n)
      v_needed += n - v[i];
  for (int i = 0; i < 6; i++)
    if (c[i] < n)
      c_needed += n - c[i];
  v_needed = max(0, v_needed - wildv);
  c_needed = max(0, c_needed - wildc);
  return v_needed + c_needed <= wildany;
}

int main() {
  //      W       I       L       D       C       A       T       S
  cin >> c[0] >> v[0] >> c[1] >> c[2] >> c[3] >> v[1] >> c[4] >> c[5];
  cin >> wildv >> wildc >> wildany;

  // binary search for the max value of n that makes can_make(n) true
  // 11111[1]00000
  int lo = 0, hi = 100000001;
  while (lo < hi) {
    // mid = (lo + hi)/2 is prone to overflow compared to mid = lo + (hi - lo)/2,
    // we also add 1, forcing division to round up, to prevent a nasty bug where
    // a search space of [no][yes] yields an infinite loop (see topcoder tutorial).
    int mid = lo + (hi - lo + 1)/2;
    if (can_make(mid)) {
      lo = mid;
    } else {
      hi = mid - 1;
    }
  }
  cout << lo << endl;
  return 0;
}
```

Problem S3: Contest Sites

This is a graph theory problem that all comes down to case work. It is good practice for many contest problems where we can make life easier by pruning out all of the trivial cases first before making bigger decisions.

The network of towns and roads can be represented as a directed, weighted graph. First we will need to determine the shortest distance from each town to each of the two contest sites. Finding shortest paths is a well-known problem in computer science, and many great online tutorials can be easily found. Two algorithms for finding the pairwise distance between every node in the graph are the Floyd-Warshall algorithm and Dijkstra's algorithm. The former is very simple to code and runs in$O(N^3)$ time while the latter will run for $O(M \log N)$ per starting node, or $O(NM \log N)$ if we run it starting from all N nodes. However, since N is as big as 10^5, both of these methods will not run in time for larger cases. To obtain full marks, we can simply reverse the directions of all edges in the graph, and then run Dijkstra's algorithm backwards from the two contest sites. This will give us the distances from every node to the contest sites in merely $O(M \log N)$ time.

Now that we have the distances from every town to the contest sites, we can indiviudally consider each town i with more than 0 competitors. Let's say for a town, the distances to the main site is D_1 and the distance to the secondary site is D_2. There are 4 possible cases:

- $D_1 = D_2 =$ infinity. This means we cannot reach either of the contest sites from the town we're currently examining. Immediately, we know that there is no way to distribute all the contestants.
- Only $D_1 =$ infinity. Since we cannot reach town 1, we know that we must send the competitors from this town to D_2. This is yet another trivial case we can get out of the way.
- $D_1 \le D_2$. Since the main site is as close or closer to the secondary site, and sending competitors has no downside, we might as well send everyone from this town to the main site.
- $D_1 > D_2$. This is the toughest case because we have to make a decision. It requires less distance for competitors in this town to attend the secondary site, but not necessarily all of them can do it considering the limit of K. We know for every competitor in this town that attends the main site over the secondary site, they will have to travel an excess distance of $D_1 - D_2$ compared to those who attend the secondary site. The solution is to sort all of the towns by decreasing value of $D_1 - D_2$ (the sacrifice to attend the main site over the secondary), and send as many competitors from the beginning of this sorted list as possible to the secondary site, until we run out of capacity there. Then we send the remainder to the primary site.

Official Solution (C++)

```
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

const int MAXN = 100005, INF = 1000000000;

int N, M, K, a, b, d, d2, C[MAXN], dist[3][MAXN];
vector< pair<int, int> > adj[MAXN]; //*reversed* adjacency list!

int main() {
  cin >> N >> M >> K;
  for (int i = 1; i <= N; i++) cin >> C[i];
  for (int i = 0; i < M; i++) {
    cin >> a >> b >> d;
    adj[b].push_back(make_pair(a, d)); //note: in reverse!
  }
  for (int i = 1; i <= N; i++)
    dist[1][i] = dist[2][i] = INF;
  //Dijkstra from main site, then from 2ndary site
  for (int start = 1; start <= 2; start++) {
    dist[start][start] = 0;
    priority_queue< pair<int, int> > q;
    q.push(make_pair(0, start));
    while (!q.empty()) {
      d = -q.top().first;
      a = q.top().second;
      q.pop();
      if (d != dist[start][a]) continue;
      for (int j = 0; j < (int)adj[a].size(); j++) {
        b = adj[a][j].first;
        d2 = adj[a][j].second + d;
        if (d2 < dist[start][b]) {
          q.push(make_pair(-d2, b));
          dist[start][b] = d2;
        }
      }
    }
  }
  vector< pair<int, int> > v;
  long long ans = 0;
  for (int i = 1; i <= N; i++) {
    if (C[i] == 0) continue;
    if (dist[1][i] >= INF && dist[2][i] >= INF) {
      cout << "-1" << endl; //we cannot reach 1 or 2 from node i
      return 0;
    }
    if (dist[1][i] <= dist[2][i]) {
      //might as well attend main site, since there's no limits
      ans += (long long)C[i] * dist[1][i];
    } else if (dist[1][i] >= INF) {
      //main site not reachable - we must attend the 2ndary site
      ans += (long long)C[i] * dist[2][i];
      K -= C[i];
    } else { // dist[1][i] > dist[1][i]
      //we have to make a decision here later.
      v.push_back(make_pair(dist[1][i] - dist[2][i], i));
    }
  }
  if (K < 0) { //we've already reached a limit on the secondary site!
    cout << "-1" << endl;
    return 0;
  }
  sort(v.rbegin(), v.rend()); //sort by decreasing dist[1][i] - dist[2][i]
  //greedily send competitors to the 2ndary site to save distance
  for (int i = 0; i < (int)v.size(); i++) {
    a = v[i].second; //the current town
    b = min(K, C[a]); //number of people from a to send to site 2
    ans += (long long)dist[1][a] * (C[a] - b); //send to site 1
    ans += (long long)dist[2][a] * b;          //send to site 2
    K -= b;
  }
  cout << ans << endl;
  return 0;
}
```

Problem S4: Chocolate Milk

Once again, we can see the system as a graph with N nodes (cisterns). Due to the fact that there are N–1 edges and no cycles, we can also see that the graph is a tree. You can read more about properties of trees from this article. In this case, we can see that cistern 1 is the root of the tree, and that cistern C_i is the parent of node i. In addition, the problem conveniently guarantees that node i is the parent of node j if and only if $i < j$. Everything about it points towards a solution using the classic dynamic programming (DP) on a tree approach. Many articles for DP on a tree can be found with a quick search online.

Let our dynamic programming state DP[i][j] represent the maximum amount of chocolate milk which can flow into cistern i (from all sources, direct or indirect), after j of the pipes in i's subtree (not including the pipe leading down from it, if any) have been upgraded. The final answer (the maximum amount of chocolate milk which can flow into cistern 1 after K pipes have been upgraded) will then be DP[1][K].

For each cistern i from N down to 1, we'll compute DP[i][0..K] all at once based on the DP values of its children. For each of it's child c, we can choose to distrubute some number u of pipe upgrades to its subtree, and we can choose to either upgrade to pipe from u to i (in which case the amount of flow into i will be DP[c][u]), or not (in which case it'll be max{DP[c][u], F_u}). If $i \geq 2$, an additional flow of P_i will also always be added. We can explore all possible decisions regarding how to distribute pipe upgrades amongst i's children using secondary DP, letting DP2[j][k] be the maximum flow from the first j children with k pipe upgrades. Letting M_i be the number of children that cistern i has, we can then copy over the results into the primary DP, with DP[i][k] = DP2[M_i][k] + P_i (for $0 \leq k \leq K$).

The secondary DP takes $O(M_i \times K^2)$ time to compute for each cistern i. Note that the sum of all M_i for all i = 1..N is equal to $N - 1$, since every cistern (except the root) is the child of one other cistern, so this DP (and the whole solution) will take a total of $O(N \times K^2)$ time. The following is a recursive solution that implements this idea.

Official Solution: Recursive DP (C++)

```
#include <iostream>
#include <cstring>
#include <vector>
using namespace std;

const int MAXN = 205;

int N, K, P[MAXN], C, F;
int DP[MAXN][MAXN], DP2[MAXN][MAXN];
vector< pair<int, int> > adj[MAXN];

void rec(int i, int prev) {
  for (int j = 0; j < (int)adj[i].size(); j++)
    if (adj[i][j].first != prev)
      rec(adj[i][j].first, i);
  memset(DP2, 0, sizeof DP2);
  int n = 1;
  for (int j = 0; j < (int)adj[i].size(); j++) {
    if (adj[i][j].first == prev) continue;
    int f = adj[i][j].second;
    n++;
    for (int a = 0; a <= K; a++) {
      for (int b = 0; b <= a; b++) {
        DP2[n][a] = max(DP2[n][a],
                        DP2[n - 1][a - b] + min(f, DP[adj[i][j].first][b]));
        if (b > 0)
          DP2[n][a] = max(DP2[n][a],
                          DP2[n - 1][a - b] + DP[adj[i][j].first][b - 1]);
      }
    }
  }
  for (int j = 0; j <= K; j++)
    DP[i][j] = DP2[n][j] + P[i];
}

int main() {
  cin >> N >> K;
  for (int i = 2; i <= N; i++) {
    cin >> P[i] >> C >> F;
    adj[C].push_back(make_pair(i, F));
  }
  rec(1, -1);
  cout << DP[1][K] << endl;
  return 0;
}
```

A more concise bottom-up approach using a similar idea is implemented as follows. How it is adapted from from the solution above is left as an exercise for the reader.

Official Solution: Non-recursive DP (C++)

```
#include <iostream>
using namespace std;

const int MAXN = 205;

int N, K, P[MAXN], C[MAXN], F[MAXN];
int DP[MAXN][MAXN], DP2[MAXN][MAXN];

int main() {
  cin >> N >> K;
  for (int i = 2; i <= N; i++)
    cin >> P[i] >> C[i] >> F[i];
  for (int i = N; i >= 1; i--) {
    int n = C[i];
    for (int j = 0; j <= K; j++) {
      for (int k = 0; k <= K - j; k++)
        DP2[n][j + k] = max(DP2[n][j + k],
                            DP[n][k] + min(F[i], P[i] + DP[i][j]));
      for (int k = 1; k <= K - j + 1; k++)
        DP2[n][j + k] = max(DP2[n][j + k],
                            DP[n][k - 1] + P[i] + DP[i][j]);
    }
    for (int j = 0; j <= K; j++) {
      DP[n][j] = max(DP[n][j], DP2[n][j]);
    }
  }
  int ans = 0;
  for (int i = 0; i <= N; i++) ans = max(ans, DP[1][i]);
  cout << ans << endl;
  return 0;
}
```